Securing Resource-Constrained Processors with Name Confusion

Mohamed Tarek Ibn Ziad
Columbia University
mtarek@cs.columbia.edu

Miguel A. Arroyo
Columbia University
miguel@cs.columbia.edu

Evgeny Manzhosov
Columbia University
evgeny@cs.columbia.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Simha Sethumadhavan
Columbia University
simha@cs.columbia.edu

***Abstract*—We introduce a novel concept, called Name Confusion, and demonstrate how it can be employed to enhance the security of resource-constrained processors. By building upon Name Confusion, we derive Phantom Name System (PNS): a security protocol that provides multiple names (addresses) to program instructions. Unlike the conventional model of virtual memory with a one-to-one mapping between instructions and virtual memory addresses, PNS creates N mappings for the same instruction, and randomly switches between them at runtime. PNS achieves fast randomization, at the granularity of basic blocks, which mitigates certain classes of code-reuse attacks.**

If an attacker uses a memory safety-related vulnerability to cause any of the instruction addresses to be different from the one chosen during a fetch, the exploited program will crash. We quantitatively evaluate how PNS mitigates real-world code-reuse attacks by reducing the success probability of typical exploits to approximately 10^{-12}. We implement PNS and validate it by running the SPEC CPU2017 benchmark suite on Gem5. Our evaluation results show that PNS has negligible performance overhead, compared to commercially-available hardware-based protections. Due to its simple design, PNS can have other use cases beyond mitigating code-reuse attacks.

I. Introduction

Virtual memory addresses serve as references, or *names*, to objects (i.e., instructions, data) during computation. For instance, every instruction in a program is uniquely identified (at run time) with a virtual memory address: the value in the Program Counter (PC). Typically, the virtual memory address assigned to an instruction is kept constant and unique for the life time of the program. In this work, we show that having multiple names for an instruction—at any given time instant—improves the security of the system with minimal hardware support without performance degradation.

How can having multiple names improve security? Given multiple names for an instruction, we define a *security protocol* that specifies a random sequence of names to be used during execution. If the attacker does not follow the security protocol by supplying an incorrect name, the exploited program will crash. In other words, if there are N addresses (names) per instruction, and if the attacker has to reuse P instruction sequences to complete an attack, the probability of detecting the attack is $1-(1/N)^P$, without any false positives. For example, for $N = 256$ and $P = 5$, then the probability of an attack succeeding is 1 in 1 trillion. This kind of protection makes this technique suitable to be used as a standalone solution, or in tandem with other, heavier-weight hardening mechanisms. We refer to such classes of architectures as *Name Confusion Architectures*.

Name confusion is fundamentally different from other hardening paradigms. For example, in the information-hiding paradigm [30], the program addresses (or parts of them) are kept a secret, but there is only one name per instruction. Similarly, Instruction Set Randomization (ISR) techniques [34], [46], [53] randomize the encoding of instructions in memory, while also maintaining a unique instruction name per program execution. In the metadata-based paradigm, such as Control-Flow Integrity (CFI) [1], [12], the set of targets (names) that can result from the execution of certain instructions (i.e., indirect branches) are computed statically and checked during execution. In moving target paradigms, such as Shuffler [65] and Morpheus [27], the names of instructions change over time; however, at any given time, there is only one valid name/address for an instruction.

In this work, we explore an application of a name confusion-based architecture, and show how it is used to mitigate a class of attacks known as code-reuse (aka return oriented programming, ROP [51], [11], [15]), including their just-in-time variants [54]. The instance we consider, called *Phantom Name System* (PNS), provides up to N different names, for any instruction, at any given time, where N is a configurable parameter (it is set to 256 in our design). The security protocol for PNS is simply a truly random selection among the different names. Specifically, PNS works as follows: during instruction fetch, the address used to fetch the instruction is randomly chosen from one of the N possible names for the instruction, and the instruction is retrieved from that address. From that point on, any PC-relative addresses used by the program relies on the name obtained during fetch. If the attacker's strategy causes any of the PC-relative addresses to be different from the one used during the fetch, then an invalid instruction will be executed, leading to unexpected effects, such as an alignment, or instruction-decoding, exception. These unexpected effects lead to program crashes that can work as signals of bad actions taking place especially in the case of repeated crashes.

Orthogonal mechanisms that turn these signals into a defensive advantage exist [41].

A naive implementation of PNS would require each instruction to be stored in N locations so they will have N names. Consequently, the capacity of all PC-indexed microarchitectural structures would be divided by N, heavily impacting performance. Further, this requires changes to the compiler, linker, loader, *etc.* In this work, we use a simple technique to avoid these problems: we intentionally *alias* the different instruction names/addresses so they point to the same instruction, allowing us to serve the N instructions from one copy. This idea is similar to how multiple virtual addresses can point to the same physical addresses (used to implement copy-on-write [9]) with two key differences: first, in PNS the N names correspond to the same virtual address, not a physical address; and second, the PNS addresses do not need to be page-aligned as required for data synonyms—i.e., PNS virtual address names can be arbitrarily offset. The first difference ensures that PNS can be handled at the application level without requiring significant changes to the operating system (OS), which manages the virtual-to-physical address mappings, while the second is key to providing security.

With the above optimizations, we show that ROP attack protection is provided at almost no performance overhead and without any binary changes. Additionally, we propose potential attacks against PNS and detail their constraints in order to guide future research. We further show, that our scheme can be combined with previously known techniques [18], [60], [43], [39] that encrypt instruction addresses stored in the heap or the global data section(s), viz., function pointers, to provide robust security against even larger class of attacks, such as JOP [6], COP [29], and COOP [50]. The combined protection scheme has 6% performance overhead, making it better than state-of-the-art commodity security solutions, like the ARM pointer authentication code (PAC) [48] that is available in the latest iPhone devices, and has the additional benefit of not requiring a 64-bit architecture. Supporting non-64-bit architectures is important as they make up the majority of the computing devices that exist nowadays: in 2019, 11.74 million servers shipped worldwide [56] vs. 25.57 billion 32-bit (or smaller) microcontrollers [57].

II. Name Confusion Architecture

A name confusion architecture assigns different addresses, or names, to any contiguous group of instructions randomly at runtime. In this section, we introduce PNS, a security protocol derived from the principles of name confusion architectures.

PNS consists of N *phantoms* (domains). It requires every instruction in the program to have N unique names. To assign the names, we use a mapping function, $name_p = f(va, p)$, which takes the instruction virtual address, va, and a phantom index, p as inputs and returns the phantom name, $name_p$. This way any instruction is mapped by f to unique location in each of the N phantoms. The function f does not have to be kept a secret, as security is purely derived from the random selection of p at fetch time. For mapping a phantom name to its original

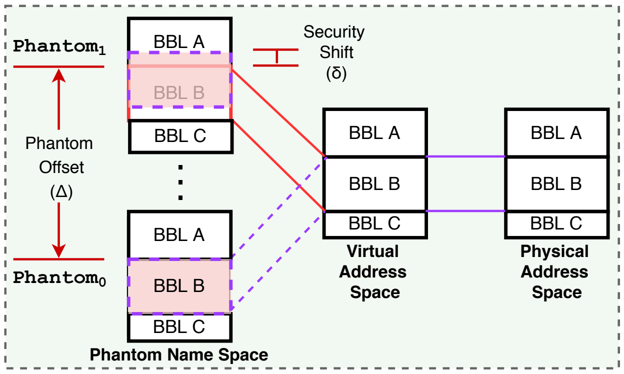

Fig. 1: Basic block mapping for PNS. BBLs are only duplicated in the phantom space.

virtual address, we use the inverse function, $va = f^{-1}(name_p)$. To enable the inverse function we ensure that the phantom name encodes the phantom index, p as part of the phantom address.

A. PNS Framework

There are four main operations to realize PNS: *Populate*, *Randomize*, *Resolve*, and *Conceal*.

Populate. PNS creates multiple phantoms of basic blocks, and populates them in the phantom name space. The left-hand side of Figure 1 shows a program with two Phantoms, such that every basic block (BBL) has two different names (addresses) in Phantom$_0$ (aka the original domain) and Phantom$_1$. PNS separates the two Phantoms by a phantom offset, Δ, in the phantom space. To add discrepancy between the Phantom copies, we introduce a minor security shift, δ, so that they are not perfectly overlapped after removing Δ. This is shown by the shaded basic block in Figure 1 and is necessary for security, as will be illustrated in Section IV-C. The inverse mapping function f^{-1} maps all phantoms to a single name in the virtual address space, which is then translated to a physical address by the OS.

Randomize. We modify the hardware to randomize program execution between the Phantoms at runtime. For example, some basic blocks will be executed from Original (Phantom$_0$) while other basic blocks will be executed from any other Phantom. Correctness is unaltered because all Phantoms provide the same functionality by construction.

Resolve. Accessing different instruction names at runtime incurs additional performance overheads as each name needs to be translated to a virtual address and then a physical one before usage. To mitigate this problem, PNS uses the inverse mapping function f^{-1} to resolve the different Phantoms to their archetype basic block. By doing so, the processor back-end continues to operate as if there is only one copy of the program in the phantom name space.

Conceal. Normal programs push return addresses to the architectural stack to help return from non-leaf function calls. The attacker may learn the domain of execution, the Phantom index, by monitoring the stack contents at runtime using arbitrary memory disclosure vulnerabilities [54]. Thus to preserve name confusion, we need to conceal the execution domain of the instructions.

B. PNS Construction

In this section, we discuss alternative design choices for the different operations in the PNS framework.

Populate. Many approaches can be used to populate the Phantoms. One approach is to use the most significant bits (MSBs) to separate the program copies in the phantom space. For example, a Δ of `0x8000_0000_0000_0000` will create two phantoms on 64-bit systems, where each phantom resides in one half of the address space. This approach is acceptable for 64-bit systems because `VA` allows for 64 bits, yet only 48 are used in practice, leaving the higher order bits available for phantom addresses. However, this is costly for 32-bit systems as it will reduce the effective range of addresses a program can use by half. Instead, to store the phantom index we add n additional bits to the hardware program counter, while maintaining the 32-bit virtual address space of the program. This allows PNS to generate $N = 2^n$ phantoms. Specifically, f, sets the additional n bits at control-flow transitions to randomize the execution at runtime. For simplicity, we set the phantom offset as $\Delta = 1 \ll 32$ and the minor security shift of any phantom to be a multiple of the phantom index (i.e., $\delta_p = p \times \delta$). We elaborate more on the PNS realization in Section III.

Randomize. PNS can randomize program addresses at any level of granularity, ranging from individual instructions to entire programs. In the rest of the paper, we use basic blocks as our elements of interest. We do not evaluate finer granularities here due to the lack of a strong security need. We define the basic block as a single entry, single exit region of code. Thus, any instruction that changes the `PC` register (referred to by control-flow instructions, such as `jmp, call, ret`) terminates a BBL and starts a new one.[1]

Conceal. We can prevent attackers from learning the execution domain in a number of ways. One straightforward way is to encrypt the return address with a secret key and only decrypt it upon function return. Another key-less, and low overhead, method that we implement is to split this information so that the public part is what is common between the phantom domains, and the private part that distinguishes the domains is hidden away without architectural access.

We split the return addresses between the architectural stack and a new hardware structure called the `Secret Domain Stack` (SDS), which by construction is immutable to external writes. SDS achieves this goal by splitting the return address $(32+n)$ bits into two parts; the n-bits, which represent the *phantom index* (p), and the lower 32 bits of the address, which encodes the *security shift* (δ). With each function `call` instruction, the lower 32 bits of the return address are pushed to the architectural (software) stack, whereas the phantom index p is pushed onto the SDS. A `ret` instruction pops the

[1]Some compilers, such as LLVM, deviate from this definition and treat `call` instructions as part of the BBL.

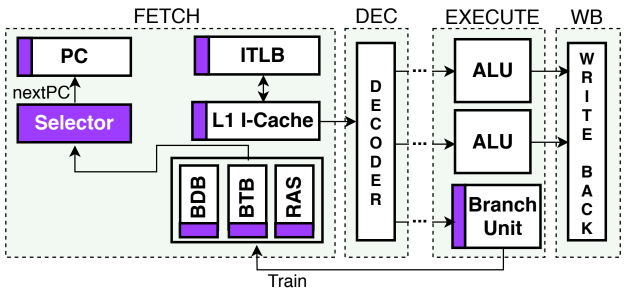

Fig. 2: Processor pipeline with PNS hardware.

most recent p from the top of SDS and concatenates it with the return address stored on the architectural stack in memory. While under attack, the return address on the architectural stack will be corrupted by the attacker. However, the attacker cannot access SDS so they cannot reliably adjust the malicious return address to correctly encode δ, leading to an incorrect target address after PNS merges the malicious return address with the phantom index p from SDS. Deployment issues with the SDS such as sizing, overflows, multithreading, *etc.* are described in Section VII.

III. Hardware Design

Figure 2 summarizes our modifications to support PNS. The changes are limited to structures that operate on `PC`.

A. Selector

With PNS, each `PC` is extended by (additional) n-bits, dubbed the *phantom index* (p). So, a program counter from phantom p will have the following format:

$$\texttt{PC}_p[31+n:0] = \{p[n-1:0], \texttt{PC}[31:0]\} \tag{1}$$

The Selector (S) is responsible for adjusting the `PC` before executing any new BBL so that the execution flow cannot be predicted by the attacker. Specifically, the selector takes the predicted target for a branch ($\texttt{PC}_{new}$) with control-flow signal s as input: s is set to one if the Branch Predictor Unit (BPU) has a predicted target for this instruction, or to zero otherwise. The selector generates the `nextPC` as the output. If s equals one, the selector generates an n-bit random phantom index p_{next}.[2] Based on p_{next}, the selector adjusts the `nextPC` according to Equation 2.

$$\begin{aligned}\texttt{nextPC}[31+n:0] = \{&p_{next}[n-1:0],\\ &\texttt{PC}_{new} - (p_{next} - p_{new}) \times \delta\}\end{aligned} \tag{2}$$

Note that p_{new} is the phantom index of the predicted target $\texttt{PC}_{new}$. For example, assuming $n = 8$-bits, we have $2^8 = 256$ phantoms. If $\texttt{PC}_{new}$ corresponds to the fifth phantom (i.e., $p_{new} = 5$) and the selector randomly chooses the eighth phantom (i.e., $p_{next} = 8$), `nextPC` will equal $\{8, \texttt{PC}_{new} - 3\delta\}$. On the other hand, if the selector randomly chooses the second phantom (i.e., $p_{next} = 2$), `nextPC` will equal $\{2, \texttt{PC}_{new} + 3\delta\}$. As the security shift δ is only used to break the overlapping

[2]This can be implemented using n metastable flip-flops [35].

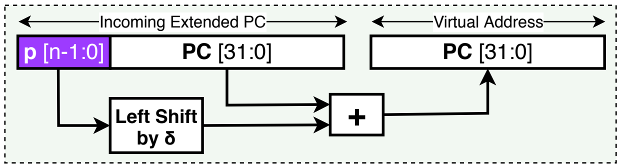

Fig. 3: Mapping the extended PC (i.e., the phantom name) to the virtual address before indexing into the microarchitectural structures.

between the names in different phantoms, it can be arbitrarily set to a single byte on CISC architectures or multiples of the instruction size on RISC architectures.

Performance Optimization #1**.** The aforementioned selector adds one cycle latency to the `nextPC` calculations in the fetch stage. To alleviate this, *we move the selector to the commit stage*—placing the selector at the commit stage allows us to mask latency overheads needed for target address adjustments so that it does not affect performance.

At the commit stage, the target of the branch instruction is known and sent back to the fetch stage to update (`train`) the BPU buffers. At this point, the selector will adjust the target address by using p_{next}, as explained above and update the BPU buffers with `nextPC`. This ensures that the next execution for this control-flow instruction will be random and unpredicted. To bootstrap the first execution of a control-flow instruction, we consider the two possible cases: correct and incorrect prediction. If the first occurrence of the control-flow instruction is correctly predicted to be `PC + 4` (falling through), then the selector will keep using the current domain of execution (unknown to the attacker) for the next BBL. If the first occurrence of the control-flow instruction is incorrectly predicted, it would be detected later on in the commit stage and the pipeline will be flushed. In this case, the selector will adjust the resolved target address by using p (unknown to the attacker) and update the BPU buffers with `nextPC`.

B. Branch Prediction Unit (BPU)

The branch prediction unit stores a record of previous target addresses in the branch target buffer (BTB), and the recent return addresses in the return address stack (RAS). For the current `PC` value, the BPU checks if the corresponding entry exists in the BTB by indexing with the PC. If it exists, the found target address becomes the `nextPC`. Otherwise, `nextPC` is incremented to `PC + 4` (or `PC + Instruction size`). If the predicted target address turns out to be incorrect later in the instruction pipeline, the processor re-fetches the instruction with the correct target address (available usually at the execute stage of the branch instruction) and nullifies the instructions fetched with the predicted target address.

Performance Optimization #2. PNS assigns N different addresses for the same control-flow instruction. In this case, we will have multiple entries in the prediction tables for the same effective instruction; this reduces the capacity to $\frac{100}{N}$%. To handle this issue, *we map the incoming phantom address to its original name before indexing into the BPU tables*, as shown in Figure 3. We do so by modifying the hashing function of the BPU tables to avoid adding any latency to the lookup operation. This way we guarantee that all phantom addresses (names) map to the same table entry. After indexing, we get the desired values from the prediction tables. As explained in Section III-A, the `nextPC` values stored in the BTB are already chosen at random from the last successful commit of this control-flow instruction (or any of its phantoms). The branch direction prediction results (Taken vs. Not Taken) in the branch direction buffer (BDB) remain the same.

C. Translation Look-aside Buffer (TLB)

Performance Optimization #3**.** Similar to the BPU buffers, the fact that we have N variants of every BBL with different virtual addresses may lead to multiple different virtual-to-physical address entries in the TLB for the same translation, reducing its capacity to $\frac{100}{N}$%. To avoid potential performance degradation, *we map the incoming phantom address to its original name before accessing the ITLB*. For example, the following two phantom addresses, `{2, 0x00BB_FFF4}` and `{0, 0x00BB_FFF8}`, will point to the same virtual address, `0x00BB_FFF8`. This common virtual address has a unique mapping to a physical address, `0x0011_DDFC`, that is stored in the ITLB. Thus, the translations related to all Phantoms map to a single entry in the ITLB, while we do not modify physical addresses so that the stored physical address part of the translation remains unaffected.

D. Instruction Cache

Performance Optimization #4**.** Creating N variants of the code sections for each program means that the L1-I\$ capacity would be effectively reduced to $\frac{100}{N}$%. PNS maps the incoming phantom address to its virtual address before accessing the L1-I\$ (in case of virtually-indexed caches) or performing the tag comparison (in case of virtually-tagged caches).[3] This represents our simple inverse mapping function, f^{-1}. The latency of the adjustment operations (shifting and addition) can be masked within the cache read operation. This incoming address adjustment ensures that while executing a $\text{BBL}_{Phantom}$ we fetch the correct instruction.

E. Execution Unit

Performance Optimization #5**.** If the target architecture allows forwarding the `PC` register through the pipeline for regular instructions, we make sure that *the `PC` register is always mapped to the virtual address before operating on it*. This mapping may introduce additional latency for the execute stage as it should be done *before/after* it. To mask such latencies, one solution is to always forward the two versions, Phantom_p and Original, of the `PC` register to the desired execution units. Although such a solution completely hides the adjustment latency, it may increase the execution unit(s) area.

[3]No changes are needed for Physically-Indexed Physically-Tagged (PIPT) caches.

F. *Secret Domain Stack*

Performance Optimization #6. Unlike prior work, which stores a complete version of the return addresses (e.g., 32-bit on `AARCH32`) in what is called a shadow stack [14], we only store $n = 8$ bits per return address. To minimize silicon area within the processor and facilitate managing the SDS, as discussed in Section II-B, we do not need to store the full return address. This structure does not introduce additional latency as it is accessed in parallel to the normal architectural stack access. We evaluate the optimal size of SDS in Section VI.

IV. Code Reuse Protection with PNS

Here, we summarize code-reuse attacks (CRAs) and defenses, and discuss how PNS is used to mitigate such attacks.

A. *Background*

Attacks that chain together gadgets whose last instruction is a `ret` are known as return oriented programming (ROP) attacks [51], [11]. ROP attacks typically start by analyzing the victim program to identify the code gadgets, which are sequences of instructions that end with a return. Afterwards, a memory corruption vulnerability is used to inject a sequence of return addresses corresponding to a sequence of gadgets. When the function returns, it returns to the location of the first gadget. As that gadget terminates with a return, the control-flow will transfer to the next gadget and so on. As ROP executes legitimate instructions belonging to the program, it is not prevented by W^X [22]. Note that variants of ROP that use indirect `jmp` or `call` instructions, instead of `ret`, to chain the execution of small instruction sequences together also exist, dubbed jump-oriented programming (JOP) [6] and call-oriented programming (COP) [29], respectively.

B. *Currently Deployed Mitigations*

The standard mitigation technique against ROP attacks is address space layout randomization (ASLR), which is currently a well-adopted defense, enabled on (pretty much) every contemporary OS [63]. Essentially, ASLR forces the attacker to first *disclose* the code layout (e.g., via a code pointer) to determine the addresses of gadgets. Snow *et al.* [54] observed that typical programs have multiple memory disclosure vulnerabilities. They developed a just-in-time ROP (JIT-ROP) compiler that explores the program's memory, disassembling any code it finds (in memory), as well as, searching for API/system calls. Then, they construct a compatible code-reuse payload *on the fly*. Note that, in principle, JIT-ROP is not restricted to dynamically stitching together only ROP payloads; it can also compile JOP, COP, or any other code-reuse payload.

Recently, ARM introduced PAC in `Armv8.3A`, which is implemented in the Apple's iPhone XS SoC [48]. The idea is based on a concept known as cryptographic control-flow integrity (CCFI) [43]. For every code pointer, such as return addresses and function pointers, CCFI stores a cryptographically-secure authentication code in the pointer's unused most significant bits. Checking the authentication code of a pointer before any indirect branches prevents control-flow hijacking because the attacker cannot compute a valid authentication code without access to keys. As we will show in Section VI, to achieve low overheads with this scheme, it is essential to have 64-bit architecture and to apply the solution to only a subset of the pointers: full-application of the idea on a 32-bit processor results in 91% overhead for SPEC CPU2017. In contrast, we want to enable security for 16, 32- and 64-bit systems, as non-64-bit systems are widely used in Internet-of-Things and Cyber Physical Systems. Thus, there is a need for new low overhead deployable solutions.

C. *PNS for CRA Protection*

PNS mitigates ROP by *ensuring that the addresses of the ROP gadgets in the gadget chain change after the chain is built*. This will result in undefined behavior of the payload (likely leading to a program crash). Consider the example in Figure 1: PNS simultaneously populates multiple (apparent) phantoms of the program code in the phantom name space; to successfully thwart the ROP gadget chain, the location of the ROP gadgets in all phantoms should be different [20].

Traditional in-place randomization techniques [47], [21] can be used to generate Phantoms. However, using an aggressive randomization approach will complicate the inverse mapping function, f^{-1}, which is responsible for recovering the archetype basic block from the different Phantoms. This will cause performance overheads with almost no additional security (beyond changing the gadget addresses in the phantom copies). PNS adopts a more efficient code layout randomization technique by introducing a security shift, δ, between the individual Phantoms, so that they are not perfectly overlapped after removing the phantom offset, Δ. This simplifies f^{-1} computations (as shown in Figure 3 and maintains code locality.

While the program is executing, PNS randomly decides which copy of the program should be executed next. Figure 4(a) shows the normal execution of a program, where `Inst 10` changes the control-flow of the program to a different BBL (starting with `Inst 71`). After the called BBL is executed, the control-flow is transmitted to the original landing point (`Inst 11` via a `ret` instruction). Figure 4(b) shows a successful CRA via ROP, in which the attacker uses a memory safety vulnerability to overwrite the return address stored on the stack and divert the control flow to `Inst 24` upon executing the `ret` instruction. Figure 4(c) shows the diversified execution of a program with PNS. For simplicity, we only show two phantoms and use a security shift, δ, sized to one instruction. Each control flow instruction can arbitrary choose to change the execution domain or not. Here, the `Randomize` operation decides to execute `Inst 71` from the Phantom domain. As the attacker cannot predict this runtime decision in advance, they provide the wrong gadget address on the stack (now shifted by δ). Thus, they will end-up executing a `WRONG` instruction, as shown in Figure 4(d). This `WRONG` instruction may belong to a different BBL or

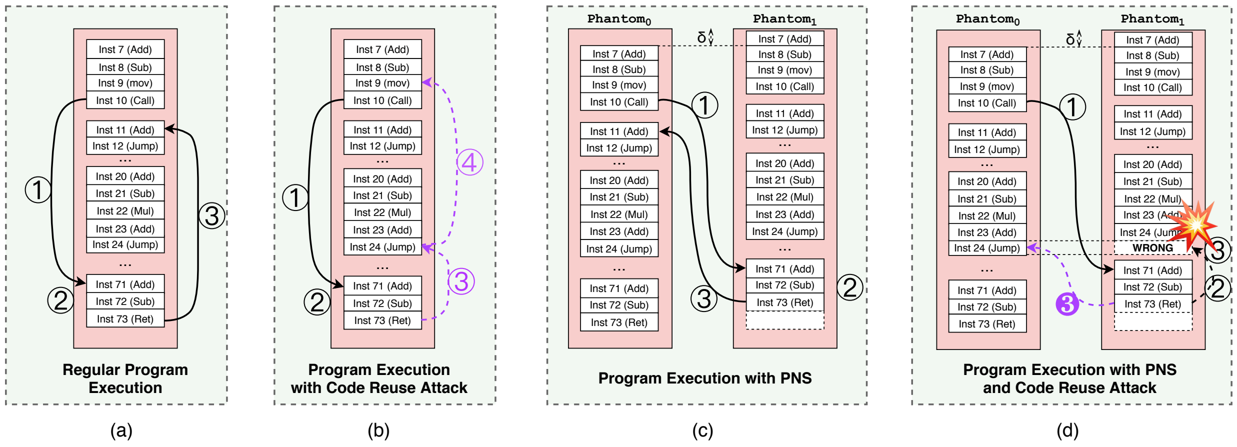

Fig. 4: PNS effect on CRAs: (a) shows the regular program execution; (b) shows successful CRA via ROP; (c) shows regular program execution with PNS; (d) shows ROP failure with PNS due to missing the desired gadget.

divert the execution to a new undesired BBL. In general, if the attacker makes the wrong guess, they will execute one less (or one more) instruction compared to the desired gadget. If δ is smaller than the instruction size, the attacker will skip a portion of the instruction resulting in an incorrect instruction decoding.

V. Security Analysis

In this section, we define the threat model of PNS and analyze its security guarantees against CRAs.

A. Threat Model

Adversarial Capabilities. We consider an adversary model that is consistent with previous work on code-reuse attacks and mitigations [43], [20], [27], [50]. We assume that the attacker has access to the source code, or binary image, of the victim program. Additionally, the victim program has one or more memory safety-related vulnerabilities that allow the attacker to read from, and write to, arbitrary memory addresses. The attacker's objective is to (ab)use memory corruption and disclosure bugs, mount a code-reuse attack, and achieve privilege escalation.

Hardening Assumptions. We assume that the underlying OS is trusted. If the OS is compromised and the attacker has kernel privileges, the attacker can execute malicious code without making ROP-style attacks; a simple mapping of the data page as executable will suffice. We assume that ASLR and WˆX protection are enabled—i.e., no code injection is allowed (non-executable data), and all code sections are non-writable (immutable code). Thus, attacks that modify program code at runtime, such as rowhammer [36], are out of scope. We also do not consider non-control data attacks [58], such as Data-Oriented Programming [31] and Block-Oriented Programming [33]. This class of attacks only tamper-with memory load and store operations, without inducing any unintended control flows in the program. This limitation also applies to prior work as well [43], [20], [12], [27]. Lastly, every other standard hardening feature (e.g., stack-smashing protection [17], CFI [12]) is *orthogonal* to PNS; our proposed scheme does not require nor preclude any such feature.

B. Security Discussion

Just-In-Time Return-Oriented Programming. Although JIT-ROP [54] permits the attacker to construct a compatible code-reuse payload *on the fly*, they cannot modify the gadget chain after the control flow has been hijacked. As a result, the attacker needs to guess the domain of execution of the entire JIT-ROP gadget-chain in advance. So, PNS mitigates JIT-ROP similarly to how it mitigates (static) ROP: i.e., by removing the attacker's ability to put together (either in advance or on the fly) a valid code-reuse payload. The above security guarantees are achieved by the regular PNS proposal (as explained in Section IV) with no extensions or program recompilation, making it suitable for legacy binaries and shared third party libraries.

Blind Return-Oriented Programming. BROP attacks can remotely find ROP gadgets, in network-facing applications, without prior knowledge of the target binary [5]. The idea is to find enough gadgets to invoke the `write` system call through trial and error; then, the target binary can be copied from memory to the network to find more gadgets. As a proof of concept, the authors showed an example with 5-gadgets that invokes `write`. With PNS, the success probability of invoking `write` would be $\left(\frac{1}{256}\right)^5 = 9.09 \times 10^{-13}$. Note that completing an end-to-end attack requires harvesting, and using, even more gadgets, after dumping the target binary, which makes the attack unfeasible on a PNS-hardened system. Additionally, BROP requires services that restart after a crash, while failed attempts will be noticeable to a system admin.

Pointer Corruption Attacks. Besides ROP, CRA variants also extensively rely on pointer corruption (e.g, JOP/COOP [6], [50]) to subvert a program's intended control flow. There also exist many software-based mitigations for JOP/COOP-like attacks [40], [66], [8], [13]. In this paper, we use a hardware-based technique for hardening PNS against them. Since the attacker needs to overwrite legitimate pointers used by indirect branches to launch the attack, we encrypt the contents of the pointer upon creation and only decrypt it upon usage (at a call site). Consequently, attackers cannot correctly overwrite it.

To achieve the above goal our Lightweight Pointer Encryption (PtrEncLite) extension adds two new instructions: `ENCP` and `DECP`. The two instructions can either be emitted by the compiler (if re-compiling the program is possible) or inserted by a binary rewriter.

- **Encrypt Pointer** (`ENCP RegX`)**.** The mnemonic `ENCP` indicates an encryption instruction. `RegX` is the register containing the pointer, e.g., virtual function pointers. The register that holds the encryption key is hardware-based and never appears in the program binary.
- **Decrypt Pointer** (`DECP RegX`)**.** The mnemonic `DECP` indicates a decryption instruction. `RegX` is the register containing the pointer. The register that holds the decryption key is hardware-based and does not appear in the program binary. As a result, the attacker cannot directly leak the key's value.

The attacker cannot simply use the above instructions as signing gadgets to encrypt/decrypt arbitrary pointers as they will have to hijack the control flow of the program first. Unlike prior pointer encryption solutions, which use weak XOR-based encryption [18], [60], PNS relies on strong cryptography (The QARMA Block Cipher Family [2]). In contrast to full CCFI solutions [43], [48], which use pointer authentication to protect all code pointers including return addresses, our approach only guards pointer usages (loads and stores). Return addresses are handled by PNS randomization, reducing the overall performance overheads, as will be shown in Section VI.

Side-channel Attacks. PNS takes multiple steps to be resilient to side channel attacks. Firstly, PNS purposefully avoids timing variances introduced due to hardware modifications, in order to limit timing-based side channel attacks. Additionally, the attacker cannot leak the random phantom index, p, which are generated by the selector as it is unreadable from both user and kernel mode—it exists within the processor only. Similarly, the execution domain cannot be leaked to the attacker through the architectural stack, as PNS keeps it within the hardware in the SDS.

C. Limitations

Whole-function Reuse. Unlike ROP attacks, which (re)use short instruction sequences, entire functions are invoked, in this case, to manipulate the control-flow of the program. This type of attack includes counterfeit object-oriented programming (COOP) attacks, in which whole C++ functions are invoked through code pointers in read-only memory, such as `vtables` [50]. PNS relies on the PtrEncLite extension to prevent the attacker from manipulating pointers (`vptr`) that point to `vtables`—a necessary step for mounting a COOP attack.

`Ret2libc` is another example for whole function reuse attacks, in which the attacker tries to execute entire `libc` functions [55], [45].[4] With PNS, the attacker will have to guess the address of the first basic block of the function in order to lunch the attack, reducing the success probability to $\left(\frac{1}{256}\right) = 0.0039$.

Our analysis of real-world exploits shows that executing a `ret2libc` attack incurs multiple steps in order for the attacker to (1) prepare the function arguments based on the calling convention, (2) jump to the desired function entry, (3) silence any side-effects that occur due to executing the whole function, and (4) reliably continue (or gracefully terminate) the victim program without noticeable crashes. (1) and (3) generally requires code-reuse (ROP) gadgets, as demonstrated by the following publicly-available exploits: (a) ROP + `ret2libc`-based exploit against mcrypt [24], (b) ROP + `ret2libc`-based exploit against Nginx [26], (c) ROP + `ret2libc` + shellcode-based exploit for Apache + PHP [23] and (d) ROP + `ret2libc`-based exploit against Netperf [25]. Thus, if the ROP part of the exploit requires G gadgets, the probability for successfully exploiting the program would exponentially decrease to $p_{success} \leq \left(\frac{1}{256}\right)^G$. That is because the attacker will have to guess the domain of execution (out of $2^8 = 256$ phantoms) of every gadget.

Repeated Observation Attacks. A potential JIT-like attack against PNS itself is what we refer to by repeated observation attack. An attacker, who can repeatedly read the architectural stack (e.g., by using a memory safety vulnerability), may record the phantomized return addresses and compare them to plaintext return addresses (i.e., return addresses that are obtained by running the same binary on a non-protected system). In this case, the attacker can recover the security shift, δ of a particular return address as the mapping function f is linear and non-secret. The attacker can then apply the observed security shift to their own malicious return address before using another memory safety vulnerability to write it to the architectural stack. While we acknowledge this hypothetical attack, it does have its own limitations that affect its practicality. For example, in addition to the above procedures, the total length of the attacker's gadget chain will be limited to the call depth at the starting point of the attack (i.e., the current depth of the SDS) as using more gadgets will cause an exception due to removing elements from an empty SDS.

VI. Evaluation

In this section, we first describe our experimental setup for evaluating PNS and its PtrEncLite extension. Then, we compare the performance of PNS against prior solutions. Finally, we quantify the security guarantees of PNS and its hardware overheads.

As we focus on resource-constrained devices, we use `ARM` ISA to demonstrate PNS as it dominates the embedded and mobile markets with its 32-bit `ARMv5-8` instruction set architecture (ISA). However, the concept of PNS can be applied to any other ISA (e.g., RISC-V).

[4]In general, any function, of any other shared library, or even the main binary itself, can be used instead.

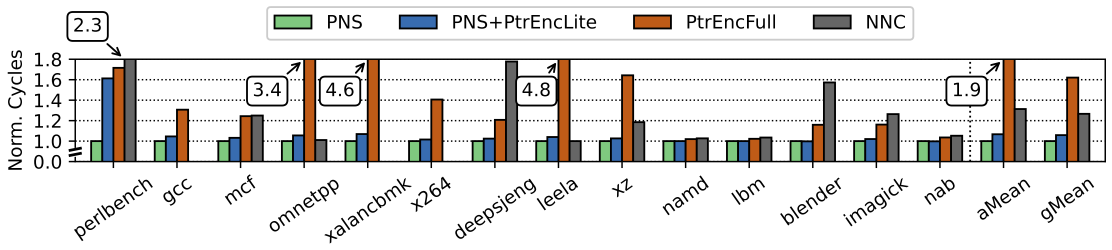

Fig. 5: PNS performance evaluation for the SPEC CPU2017 C/C++ benchmarks.[6]

Core	ARMv7a OoO core at 1.8 GHz BPred: BiModeBP, 4096-entry BTB, 48-entry RAS Fetch: 3 wide, 48-entry IQ Issue: 8 wide, 60-entry ROB Writeback: 8 wide, 16-entry LQ, 16-entry SQ
L1 I-cache	32KB, 2-way, 2 cycles, 64B blocks, LRU replacement, 2 MSHRs, no prefetch
L1 D-cache	32KB, 2-way, 2 cycles, 64B blocks, LRU replacement, 16-entry write buffer, 6 MSHRs, no prefetch
L2 cache	2MB, 16-way, 15 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 16 MSHRs, stride prefetch
DRAM	LPDDR3, 1600 MHz, 1GB, 15ns CAS latency and row precharge, 42ns RAS latency

TABLE I: Simulation parameters.

A. *Experimental Setup*

We implement PNS in the out-of-order (OoO) CPU model of Gem5 [4] for the ARM architecture. We execute ARM32 binaries from the SPEC CPU2017 [10] C/C++ benchmark suite on the modified simulator in syscall emulation mode with the ex5_big configuration (see Table I), which is based on the ARM Cortex-A15 32-bit processor.

To compile the benchmarks, we build a complete toolchain based on a modified Clang/LLVM v7.0.0 compiler including musl [44], compiler-rt, libunwind, libcxxabi, and libcxx. Using a full toolchain allows us to instrument all binary code including shared libraries and remove them from the trusted code base (TCB). In order to evaluate PNS, we use our modified toolchain to generate the following variants.

Baseline. This is the case of an unmodified unprotected machine. Specifically, we compile and run the SPEC CPU2017 benchmarks using an unmodified version of the toolchain and Gem5 simulator. In all of our experiments, we use the total number of cycles (numCycles) to complete the program, as reported by Gem5, to report performance. The numCycles values of the defenses are normalized to this baseline implementation without defenses; thus, a normalized value greater than one indicates higher performance overheads.

PNS. In this scenario, we run unmodified binaries on our modified Gem5 implementation with all optimizations, as described in Section III.

PNS-PtrEncLite. To evaluate the performance of PNS with PtrEncLite, we first write an LLVM IR pass to instrument the code (including shared libraries) and insert the relevant instructions as described in CCFI [43]. Specifically, we emit instructions whenever (1) a new object is created (to encrypt the contents of the vptr), (2) a virtual function call is made (to decrypt the vptr), or (3) any operation on code pointers in C programs. Then, we appropriate the encodings for ARM's ldc and stc instructions respectively, which are themselves unimplemented in Gem5, to behave as ENCP and DECP instructions. We add a dedicated functional unit in Gem5 to handle these instruction's latency in order to avoid any contention on the regular functional units. We also assume equal cycle counts of 8 for both instructions to emulate the effect of the actual encryption/decryption similarly to prior work [2].

PtrEncFull. In this approach, we instrument code pointer load/store operations in addition to function entry/exit points to protect return addresses for non-leaf functions. Conceptually, this solution is similar to ARM PAC [39]. However, due to the absence of PAC support in Gem5 (and for 32-bit ARM architectures in general), we only perform behavioral simulation for comparison purposes, without keeping track of the actual pointer metadata.

Naive Name Confusion (NNC). For the sake of completeness and fair comparison, we also implement a static version where there are two copies of the code, i.e., a version without the phantom aspect of the naming scheme. In this model, we have two virtual addresses for each instruction but these addresses are physically stored in memory, essentially halving the capacity of the microarchitectural structures.[5] We create the two copies by introducing a shift of TRAP instruction size in one of them. At a high-level our implementation works as follows: (1) clone functions using an LLVM IR pass, (2) LLVM backend pass to insert TRAPs for cloned functions, (3) instruct the LLVM backend to globalize BBL labels, (4) emit a diversifier BBL for every BBL, and (5) rewrite branch instruction targets to point to the diversifier.

Of the 16 C/C++ benchmarks, 14 compile with all different toolchain modifications. parest has compatibility issues with musl due to exception handling usages, while povray failed to run on Gem5. For NNC, gcc, xalancbmk, and x264 present compilation and/or linking issues.

[5]This model is similar to the Isomeron solution proposed by Davi *et al.* [20], with the modification that it is used at the BBL granularity as opposed to the original work, which uses a dynamic binary rewriting framework with function granularity.

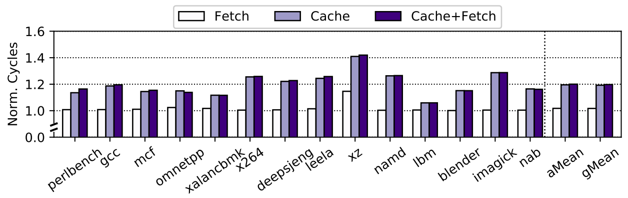

Fig. 6: PNS performance evaluation with additional one-cycle access latency for fetch stage, L1-I$, and both.

Bench. Name	Call Depth	Bench. Name	Call Depth	Bench. Name	Call Depth
perlbench	24	x264	15	lbm	10
gcc	28	deepsjeng	48	blender	23
mcf	28	leela	244	imagick	22
omnetpp	196	xz	16	nab	16
xalancbmk	77	namd	12	povray	-

TABLE II: Maximum call depth for the SPEC CPU2017 C/C++ benchmarks.

Bench. Name	$\overline{PNS}$ Chains	PNS Chains	Bench. Name	$\overline{PNS}$ Chains	PNS Chains	Bench. Name	$\overline{PNS}$ Chains	PNS Chains
perlbench	17	0	x264	23	0	lbm	23	0
gcc	23	0	deepsjeng	11	0	blender	23	0
mcf	11	0	leela	15	0	imagick	23	0
omnetpp	23	0	xz	11	0	nab	23	0
xalancbmk	15	0	namd	23	0	povray	23	0

TABLE III: ROP gadget-chain reduction for SPEC2017 C/C++ benchmarks. $\overline{PNS}$ and PNS correspond to the number of valid ROP chains before and after PNS.

B. Performance Evaluation

We run all benchmarks to completion with the `test` input set on our augmented Gem5. We verified the correctness of the outputs against the reference output. Figure 5 shows the performance overhead of the different design approaches (all normalized to Baseline). As expected, PNS has identical performance to Baseline. Adding support for PtrEncLite increases the performance overheads of PNS-PtrEncLite to 0%– 61% (avg. 6%). The `perlbench` benchmark suffers from a relatively high overhead due to its extensive use of function pointers and indirect branches. On the other hand, fully protecting the binaries with a deterministic defense such as PtrEncFull encounters a 91% overhead on average (geometric mean of 62%). Our static implementation of software NNC introduces an arithmetic average overhead of 31% (geometric mean of 26%)[6]. In contrast to software Isomeron [20] which relies on dynamic binary instrumentation (DBI), the overheads for our implementation are primarily attributed to the indirection every BBL branch must make to the diversifier.

As illustrated in Section III, the required PNS modifications do not add additional cycle latency to the processor pipeline. However, we performed an additional set of experiments with a more conservative assumption of having one additional cycle latency for all instructions in fetch stage, or one more cycle for accessing L1 instruction cache, or both. We show results compared to an unmodified baseline in Figure 6. We notice an average performance overhead of 1% for stalling the fetch stage. However, stalling the instruction cache for one cycle (hit latency is originally two cycles) is more harmful to the performance. Thus, the I$ optimizations are mandatory, as described in Section III.

Finally, the call depths listed in Table II show that SPEC programs do not exceed a depth of 244 (`leela`), indicating that a 256-entry hardware `Secret Domain Stack` is sufficient to handle the common execution cases.

C. Security Evaluation

ROP-Gadget Chain Evaluation. To evaluate PNS against real-world ROP attacks we use Ropper [49], a tool that can find gadgets and build ROP chains for a given binary. A common ROP attack is to target the `execve` function with `/bin/sh` as an input to launch a shell. As the chain-creation functionality in Ropper is only available for `x86` [49], we analyze SPEC CPU2017 `x86` binaries for this particular exploit and report the number of available gadget chains ($\overline{PNS}$).

To emulate the effect of PNS, we modified the Ropper code to extend each gadget length by one byte, decode the gadget, and check if the new gadget is semantically equivalent to the old one or not. This emulates the effect of an attacker targeting a particular address, but instead executing the one before due to the PNS security shift, δ. As shown in Table III, PNS foils all the gadget-chains found by our modified Ropper. Extending the Ropper chain-creation functionality to the ARM ISA is part of our future work. Intuitively, the results would be even worse for the attacker in ARM as the state-space is more constrained due to instruction alignment requirements.

Control-flow Hijacking Evaluation. We further evaluate security by using RIPE [64], an open source intrusion prevention benchmark suite. We port RIPE to ARM and run it on our modified Gem5, with $n = 8$ bits, as described in Section VI. We mainly focus on return-address manipulation as a target code pointer and `ret2libc`/ROP as attack payloads. Shellcode attacks are not considered as we expect W^X.

Our ported RIPE benchmark contains 54 (relevant) attack combinations. On an unprotected Gem5 system, 50 attacks succeed and 4 attacks fail. After deploying PNS, all of the 54 attacks fail including the single-gadget `ret2libc` attacks. That is mainly due to our high number of phantoms present at runtime, $2^8 = 256$.

That said, real-world exploits typically involve payloads with several gadgets. According to Cheng *et al.* [16] the shortest gadget chain consists of thirteen gadgets. Hence, the probability for successful execution of a gadget chain is $p_{success} \leq \left(\frac{1}{256}\right)^{13} = 4.93 \times 10^{-32}$. Snow *et al.* [54] successfully exploited a vulnerability with a ROP payload consisting of only six gadgets, which would equate to a better, but still low, success probability of $p_{success} = \left(\frac{1}{256}\right)^{6} = 3.55 \times 10^{-15}$.

[6]Our NNC implementation does not instrument external libraries (only the main application code) due to compilation issues. This leads to overheads that are less than intuitively expected.

D. FPGA Prototyping

For the sake of completeness, we have developed an FPGA prototype of PNS using the Bluespec hardware description language (HDL). Specifically, we added PNS hardware modifications to the front-end of the 32-bit `Flute` RISC-V processor, a 5-stage in-order pipelined processor typically used for low-end applications that need MMUs [7]. We prototyped the processor on the the Xilinx Zynq (ZCU106) Evaluation Kit.

Our evaluation results shows that we can reliably run with a clock period of 7.5 ns (maximum frequency of 133 MHz) for both the baseline core and the modified one. The area increase due to PNS is negligible (0.83% extra Flip-Flops with 2.02% additional LUTs). We verified the correctness of our FPGA implementation by running simple bare-metal applications.

VII. PNS System Level Support

For completeness, we outline design changes required to deploy a PNS general-purpose system.

Sizing. Although SDS only stores eight bits per return address in hardware, it still has a limited size that cannot be dynamically increased as the architectural stack. This means programs with deeply nested function calls may result in a SDS overflow. To handle this issue, we add two new hardware exception signals: *hardware-stack-overflow* and *hardware-stack-underflow*. The former is raised when the SDS overflows. In this case, the OS (or another trusted entity), encrypts and copies the contents of the SDS to the kernel memory. This kernel memory location will be a stack of stacks and every time a stack is full it will be appended to the previous full stack. The second exception will be raised when the SDS is empty to decrypt and page-in the last saved full-stack from kernel memory.

Stack Unwinding. Since addresses are split across the architectural (software) stack and the SDS it is vital to keep them in sync for correct operation. Earlier, we described how normal LIFO `call`/`rets` are handled. In some cases, however, the stack can be reset arbitrarily by `setjmp`/`longjmp` or C++ exception handling. To ensure the stack cannot be disclosed/manipulated maliciously during non-LIFO operations, we change the runtime to encrypt the `jmp_buffer` before storing it to memory. Additionally, we also store the current index of the SDS. When a `longjmp` is executed, we decrypt the contents of the `jmp_buffer` and use the decrypted SDS index to re-synchronize it with the architectural stack. The same approach can be applied to the C++ exception handling mechanism by instrumenting the appropriate APIs.

Context Switches. The SDS of the current process is stored in the Process Control Block before a context switch. In terms of cost, the typical size of the SDS is 256-bytes (256 entries, each has 8-bits). Moving this number of bytes between the SDS and memory during context switch requires just a few `load` and `store` instructions, which consume a few cycles. This overhead is negligible with respect to the overhead of the rest of the context switch (which happens infrequently; every tens of milliseconds).

Multithreading. To support multithreading, the SDS has to be extended with a multithreading context identifier, which increases the size of stack linearly with number of thread contexts that can be supported per hardware core.

Dynamic Linking. Dynamically-linked shared libraries are essential to modern software as they reduce program size and improve locality. Although most embedded system software (the primary target in this work) in MCUs is typically statically-linked, we note that PNS is compatible with shared libraries as it can be fully realized in hardware. Thus, it does not differentiate between BBLs related to the main program and the ones corresponding to shared libraries. On the other hand, dynamic linking has been a challenge for many CFI solutions, as control flow graph edges that span modules may be unavailable statically. CCFI [43] suffers from the same limitation as the dynamically shared library code needs to be instrumented before execution; otherwise, the respective pages will be vulnerable to code pointer manipulation attacks.

VIII. Related Work

As explained in Section I, the idea of having multiple names for the same instruction is fundamentally different compared to other security paradigms. Further, in Section VI we showed that PNS has lower overheads compared to the state-of-the-art commercial solution, ARM PAC. In this section, we explore prior CRA mitigations and discuss their benefits and differences (summarized in Table IV).

N-Variant eXecution Systems. The general idea of N-variant execution (NVX) systems is to run *N different* copies/variants of the same code, alongside each other, while checking their runtime behavior [3], [19]. If the variants produce a different response to a single common input (due to an internal failure or external attack payload), the checker detects such divergences in execution and raises an alert. Since 2006, many NVX systems have been proposed to achieve reliability and security goals [61], [62], [37], [38], [28], [42]. While NVX systems can offer additional benefits over PNS, such as precise failure detection, they suffer from considerable performance (at least 100%) and memory overheads, and therefore are not suitable for resource constrained systems.

Live Randomization. Recent work has pioneered the use of hardware moving target defenses to protect against CRAs [27]. Gallagher *et al.* proposed Morpheus, an architecture that (1) randomizes code and data pointers using relocation and strong encryption and (2) periodically repeats the first step using a different displacement and key. The main conceptual difference between Morpheus and PNS is that in PNS, at any given instant there are multiple names (addresses) for an instruction while there is only one name (address) for an instruction in Morpheus. This distinction is also true of PNS and software moving target systems [65] used to protect against CRAs.

PNS can also provide an illusion of a faster churn rate. The churn time can be thought of as the time an attacker has to deploy a countermeasure. PNS, forces the attacker to have a counter strategy every basic block which normally completes

Proposal	Hardware Support	Software Modifications	Randomization Interval	Main Sources of Overheads	Cost of Portability to 32-bit systems	Energy Overheads
NVX [3], [42]	No	Recompile	No	Running N program copies simultaneously	Increase overheads by a factor of N	High
Isomeron [20]	No	DBI	1 ms (Func. time)	Maintaining two program copies (high TLB and I$ misses)	None	High
Shuffler [65]	No	DBI	50 ms	Offloading computations to another core/thread	Double overheads on single-core systems	High
Morpheus [27]	Yes	Recompile	50 ms	Adding 2-bit tags per 64-bit words (pointer size)	Double memory tags overhead	Low
Intel CET [32]	Yes	Recompile	No	Maintaining full shadow stack	None	Low
CCFI [43]	No	Recompile	No	Using complete pointer authentication	Extra Load/Store per pointer	Moderate
ARM PAC [48]	Yes	Recompile	No	Using complete pointer authentication (negligible on h/w)	Extra Load/Store per pointer	Moderate
ZeRØ [59]	Yes	Recompile	No	Encoding pointer metadata (negligible on h/w)	Extra Load/Store per pointer	Low
PNS	Yes	None	10 ns (BBL time)	None	None	Low
PNS-PtrEncLite	Yes	Recompile	No	Using Lightweight Pointer Encryption	None	Low

TABLE IV: Comparison with prior work.

execution in the order of nanoseconds. While Morpheus' churn rate (milliseconds for PNS level of performance) is sufficient to protect against remote network adversaries, the (apparently) faster churn provided by PNS is meaningful in offering protection against local attackers especially with side channel capabilities, and thus is again complementary to Morpheus. The BBL-by-BBL apparent churn offered by PNS also comes at much lower energy cost compared to Morpheus as it does not require memory scanning to identify pointers. Finally from a deployment perspective, a unique benefit of PNS is that it works for non-64-bit systems while Morpheus and software moving target systems, rely on the availability of a 64-bit address space for security.

Hardware-based CRA Mitigations. Intel architectures offer a hardware-based CFI technology named Control-flow Enforcement Technology (CET) that is to be available in future `x86` processors [32], [52]. CET requires program recompilation in order to insert a new `ENDBRANCH` instruction at the beginning of each BBL that can be invoked via an indirect branch. At runtime, the destination of all indirect branch instructions should be an `ENDBRANCH`, otherwise an attack is assumed. CET provides only coarse-grained protection where any of the possible indirect targets are allowed at every indirect control-flow transfer. Thus, an attacker can still reuse the whole BBL and store the address of the `ENDBRANCH` of the desired BBL in the stack as before. The above attack will fail against PNS with high probability as every instruction (and basic block) can have up to N different addresses forcing the attacker to gamble on which one to use. Additionally, CET protects `call-return` instructions using a full shadow stack (i.e., 32 or 64 bits per entry), that resides in virtual memory. Unlike a shadow stack which compares return address on every `ret` instruction, our SDS only concatenates the domain bits to the return address with no wasteful comparisons. Furthermore, PNS uses a smaller hardware structure (the SDS) that consumes 8 bits per entry and that cannot be leaked by an attacker who can illegally tamper main memory.

On the other hand, ARM introduced the Pointer Authentication Code (PAC) feature in `Armv8.3A` as a hardware primitive to mitigate CRAs [48]. Hans *et al.* showed how to harden ARM PAC against reply attacks by using unique tweaks (along with the authentication key) for different pointer types [39]. As discussed in Section IV-B, ARM PAC relies on the currently unused upper bits of the 64-bit pointers. Mapping the same technique to non 64-bit systems results in high performance overheads, as evaluated in Section VI.

While our PNS-PtrEncLite extension relies on cryptographic algorithms similar to ARM PAC [48], PNS-PtrEncLite has two main advantages. First, PNS-PtrEncLite uses encryption instead of authentication to avoid storing additional metadata (authentication code) per pointer on 32-bit systems. Second, ARM PAC is applied for all code pointers including return addresses and function pointers. This is represented by PtrEnc-Full in our evaluation. On the other hand, PNS-PtrEncLite is only applied for function pointers (and C++ virtual pointers) as the return addresses are protect by PNS's fine-grained randomization. The reduction in the cryptographically-protected locations highly reduced the performance overheads, as shown in Section VI.

Recently, Ziad *et al.* proposed ZeRØ, a hardware primitive for resilient operation under pointer manipulation attacks [59]. ZeRØ uses unique instructions for accessing different categories of program pointers (i.e., return addresses, code pointers, and data pointers). At runtime, the hardware uses the unique instructions to tag program pointers and then prevents non-pointer memory accesses from manipulating them. ZeRØ's tags are currently stored in the upper bits of the 64-bit pointers using a special encoding to minimize the memory overheads. While ZeRØ provides strong security guarantees by protecting both code and data pointers, PNS neither requires 64-bit systems nor introduces any changes to the memory subsystem. Similar to PNS, ZeRØ does not require program recompilation for hardening return addresses. Compiler support is needed for code and data pointer integrity.

IX. Conclusion

In this paper, we proposed PNS, a name confusion design that allows for multiple addresses/names for individual instructions. We explored one potential application for PNS, which is mitigating code-reuse attacks. The key idea is to force the attacker to carry out the difficult task of guessing which randomly-chosen name will be used, by the hardware, to carry out a successful attack. PNS requires minor modifications to the processor front-end: specifically, it requires changes to indexing functions, 8 metastable flip-flops, and 256 bytes of state. Experimental results showed that PNS incurs negligible performance impact compared to commercially-available hardware-based solutions. Our security evaluation showed that PNS mitigates both real-world ROP exploits and synthetic benchmarks. We further illustrated how the security guarantees of PNS can be boosted when integrated with other solutions by evaluating the PNS-PtrEncLite extension. We have also discussed potential attacks against PNS and detailed their limitations.

The increased proliferation of resource-constrained systems that cannot deal with the performance overheads of server-grade defenses calls for more efficient security solutions. As PNS does not depend on "free" bits or the vastness of the 64-bit address space to work, it is a reasonable security option for 16- and 32-bit microcontrollers and microprocessors. Finally, PNS's simple hardware modifications, native performance, and support for legacy binaries hold more promise for other use cases beyond mitigating code-reuse attacks.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, Alexandria, VA, USA, 2005.

[2] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, Mar. 2017.

[3] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, Ottawa, Ontario, Canada, 2006.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[5] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, S&P '14, pages 227–242, Washington, DC, USA, 2014.

[6] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, Hong Kong, China, 2011.

[7] Bluespec. Flute: 5-stage, in-order, piplined RISC-V CPU. https://github.com/bluespec/Flute, 2019. [Online; accessed 25-April-2021].

[8] Dimitar Bounov, Rami Gokhan Kici, and Sorin Lerner. Protecting C++ dynamic dispatch through VTable interleaving. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, February 2016.

[9] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, Berlin, Germany, April 2018.

[11] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, Alexandria, Virginia, USA, 2008.

[12] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[13] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object type integrity for C++ virtual dispatch. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, USA, February 2018.

[14] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P '19, May 2019.

[15] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, Chicago, Illinois, USA, 2010.

[16] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, February 2014.

[17] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78, 1998.

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM '03, page 7, Washington, DC, USA, 2003.

[19] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Vancouver, B.C., Canada, 2006.

[20] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, USA, February 2015.

[21] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 299–310, Hangzhou, China, 2013.

[22] Ulrich Drepper. Security enhancements in redhat enterprise Linux (beside SELinux), 2004.

[23] Exploit Database. Apache 2.4.7 + php 7.0.2 - openssl_seal() uninitialized memory code execution. https://www.exploit-db.com/exploits/40142. [Online; accessed 25-April-2021].

[24] Exploit Database. mcrypt 2.5.8 - local stack overflow. https://www.exploit-db.com/exploits/22928. [Online; accessed 25-April-2021].

[25] Exploit Database. Netperf 2.6.0 - stack-based buffer overflow. https://www.exploit-db.com/exploits/46997. [Online; accessed 25-April-2021].

[26] Exploit Database. Nginx 1.3.9 ¡ 1.4.0 - chuncked encoding stack buffer overflow. https://www.exploit-db.com/exploits/25775. [Online; accessed 25-April-2021].

[27] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 469–484, Providence, RI, USA, 2019.

[28] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. Detile: Fine-grained information leak detection in script engines. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA '16, pages 322–342, San Sebastian, Spain, 2016.

[29] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, S&P '14, pages 575–589, San Jose, CA, USA, may 2014.

[30] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermin-

ing information hiding (and what to do about it). In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119, Austin, TX, August 2016.

[31] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, S&P '16, pages 969–986, San Jose, CA, USA, May 2016.

[32] Intel. Intel control-flow enforcement technology preview. https://binpwn.com/papers/control-flow-enforcement-technology-preview.pdf, 2017. [Online; accessed 25-April-2021].

[33] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, Toronto, Canada, 2018.

[34] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, Washington D.C., USA, 2003.

[35] Lee-Sup Kim and Robert W. Dutton. Metastability of CMOS latch/flip-flop. *IEEE Journal of Solid-State Circuits*, 25(4):942–951, Aug 1990.

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Minneapolis, Minnesota, USA, 2014.

[37] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '16, pages 431–442, June 2016.

[38] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. LDX: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 503–515, Atlanta, Georgia, USA, 2016.

[39] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium*, (USENIX Security 19), pages 177–194, Santa Clara, CA, USA, August 2019.

[40] LLVM. Control flow integrity design. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. [Online; accessed 25-April-2021].

[41] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Application communities: Using monoculture for dependability. In *Proceedings of the First Conference on Hot Topics in System Dependability*, HotDep'05, page 9, Yokohama, Japan, 2005. USENIX Association.

[42] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[43] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, Denver, Colorado, USA, 2015.

[44] musl. musl libc. http://www.musl-libc.org/. [Online; accessed 25-April-2021].

[45] Nergal. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html, 2001. [Online; accessed 25-April-2021].

[46] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: Architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 981–992, Berlin, Germany, 2013.

[47] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, S&P '12, pages 601–615, May 2012.

[48] Inc Qualcomm Technologies. Pointer authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017. [Online; accessed 25-April-2021].

[49] Sascha Schirra. Ropper. https://github.com/sashs/Ropper. [Online; accessed 25-April-2021].

[50] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15, pages 745–762, Oakland, CA, USA, 2015.

[51] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, Alexandria, Virginia, USA, 2007.

[52] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, Phoenix, AZ, USA, 2019.

[53] Kanad Sinha, V. P. Kemerlis, and Simha Sethumadhavan. Reviving instruction set randomization. In *Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28, 2017.

[54] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P '13, pages 574–588, Berkeley, CA, USA, May 2013.

[55] Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63, August 1997.

[56] Statista, Gartner, and IDC. Server shipments worldwide from 2010 to 2020. https://www.statista.com/statistics/219596/worldwide-server-shipments-by-vendor/, 2021. [Online; accessed 25-April-2021].

[57] Statista and IC Insights. Microcontroller unit (mcu) shipments worldwide from 2015 to 2023. https://www.statista.com/statistics/935382/worldwide-microcontroller-unit-shipments/, 2020. [Online; accessed 25-April-2021].

[58] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P '13, pages 48–62, San Francisco, CA, USA, 2013.

[59] M. Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. ZeRØ: Zero-overhead resilient operation under pointer integrity attacks. In *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, pages 999–1012, Worldwide Event, June 2021.

[60] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *37th International Symposium on Microarchitecture (MICRO '04)*, pages 209–220, Portland, OR, USA, 2004.

[61] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, July 2016.

[62] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ATC '16, pages 167–179, Denver, CO, USA, 2016.

[63] Ollie Whitehouse. An analysis of address space layout randomization on windows vista, Jan 2007.

[64] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, Orlando, Florida, USA, 2011.

[65] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 16, pages 367–382, Savannah, GA, USA, 2016. USENIX Association.

[66] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. Vtrust: Regaining trust on virtual calls. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, February 2016.